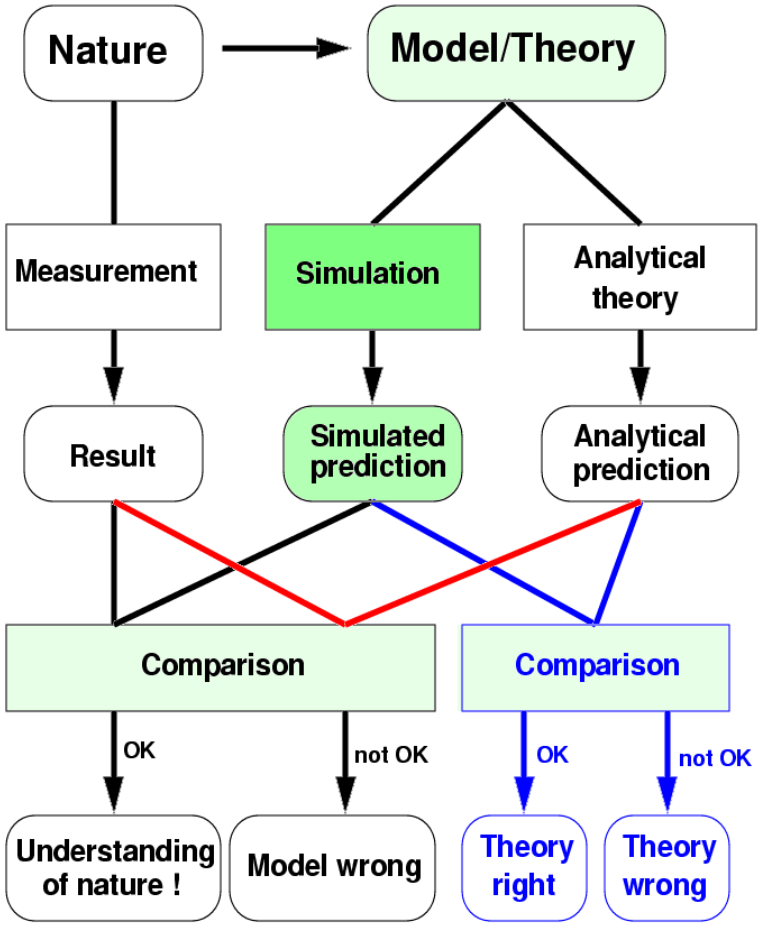
12. Repetition and final comments

Terminology: to all purposes known to the lecturer, the terms "stochastic simulations" and "Monte Carlo simulations" can be considered identical. Usage should conform to common usage in the particular subfield.

12.1. Where have we been in science methodology?

Let's repeat the schematic figure from the beginning of the course. But now I have marked in green on which parts of this graph we have been during this course. The darker the green shade in the box, the more we have dealt with that part.



Basically, we have taken as given basic physics theories, spent most of the time showing how these can be simulated, and a bit on how they can be compared to experiments and analytical theory. We have also dealt a lot with topics which are not really on this graph, such as random number

generation and MC integration. These can be considered to be computer science and mathematics topics which serve as input to the "simulation" box.

12.4. Overview of random number generation

[From chapters 4-5 on the course.]

In physics applications repeatability is usually desired. Hence in this summary we only consider deterministic random numbers.

12.4.1. The linear congruential-based generators

One of the simplest decent generators:

$$I_{j+1} = aI_j + c \qquad (\text{mod } m) \tag{1}$$

Here "mod" is the modulus (remainder of division) operation.

For well-chosen a, c and m this can be pretty good. The Park-Miller minimal generator has

$$a = 7^5 = 16807 \qquad c = 0 \qquad m = 2^{31} - 1$$

The main problem with this is that it produces stripes on a fine scale in 2D. This can be circumvented pretty well by adding a shuffle operation.

Two Park-Miller generators with a shuffle can be combined to form a really nice generator, implemented as `ran2` in Numerical Recipes. It has a period of $\approx 2.3 \times 10^{19}$.

These generators can be modified to become non-linear. The simplest trick is to rewrite the generator as

$$y_{n+1} = a\bar{y}_n + b \qquad (\text{mod } M)$$

where $\bar{y}$ signifies solving the equation

$$y_n \bar{y}_n = 1 \qquad (\text{mod } M)$$

12.4.2. Other approaches to random number generators

12.4.2.1. The GFSR generator

In the GFSR family of generators, XOR operations are used on a set of previous random numbers to generate new numbers.

In the original generator, one starts with p random integer numbers $a_i, i = 0, 1, \ldots, p$ generated somehow in advance. Then the new elements k, with $k \geq p$, can be generated as

$$a_k = a_{k-p+q} \oplus a_{k-p}$$

where p and q are constants, $p > q$, and $\oplus$ is the XOR logical operation.

For small p these generators can be terrible, but for $p > 1000$ they pass even very advanced tests. The disadvantage here is only that we need memory proportional to p.

12.4.2.2. Combined generators

Many of the generators considered among the best combine some of the simpler generator.

The `RANMAR` generator has a period of 2^{144}.

The Mersenne twister generator has a period of $2^{19937} - 1$

Both are also quite fast.

12.4.2.3. Which generator should be used

As of the writing of this (May 2004), the `ran2,` and Mersenne twister generators had not been known to fail in any test. RANMAR has failed in some theoretical tests, but is in very wide use and apparently most users are happy with it. Hence one can feel fairly much on the safe side with any of them.

However, if you write code which may be used 10 or 20 years from now, `ran2` may start to fail because of exhaustion of the period of only 2×10^{19}.

12.4.3. Generating non-uniform distributions

12.4.3.1. Analytical approach

Given a function for which

$$f(x) > 0 \text{ for all } x \quad \text{and} \quad \int_{-\infty}^{\infty} f(x)dx = 1$$

we can generate random numbers as follows.

Calculate the cumulative function

$$F(x) = \int_{-\infty}^{x} f(t)dt$$

and its inverse function $F^{-1}(s)$,

$$s = F(x) \Longleftrightarrow x = F^{-1}(s)$$

We can then get random numbers r distributed as $f(x)$ as follows:

1° Generate a uniformly distributed number $u = P_u(0, 1)$

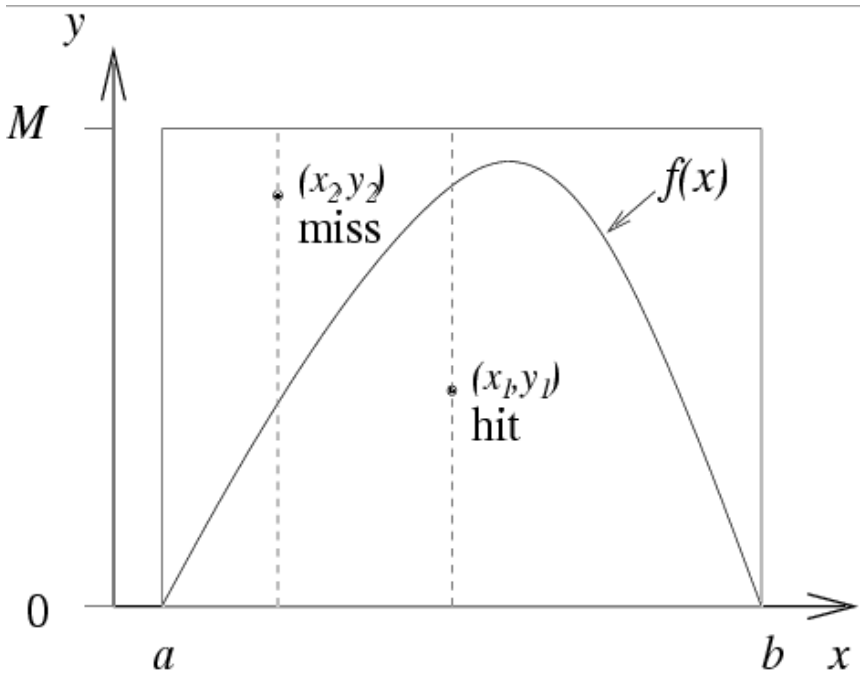
2° Calculate $x = F^{-1}(u)$

For a discrete distribution, the same thing can be easily done numerically. For good accuracy, parabolic or cubic spline interpolation may be desirable. However, if $f(x) = 0$ anywhere in the interval, one has to take care to prevent numerical errors.

For an analytical function f, in case it is not possible to find $F(x)$ or $F^{-1}(s)$, one can tabulate f numerically, then use the discrete approach.

12.4.3.2. von Neumann rejection method

Consider a function $f(x)$ defined in some finite interval $x \in [a, b]$. It has to be normalized to give probabilities. Let M be an number which is $\geq f(x)$ for any x in the interval:



Now we can generate random numbers in this distribution as

1° Generate a uniformly distributed number $x = P_u(a, b)$

2° Generate a uniformly distributed number $y = P_u(0, M)$

3° If $y > f(x)$ this is a **miss:** return to 1°

4° Otherwise this is a **hit:** return x

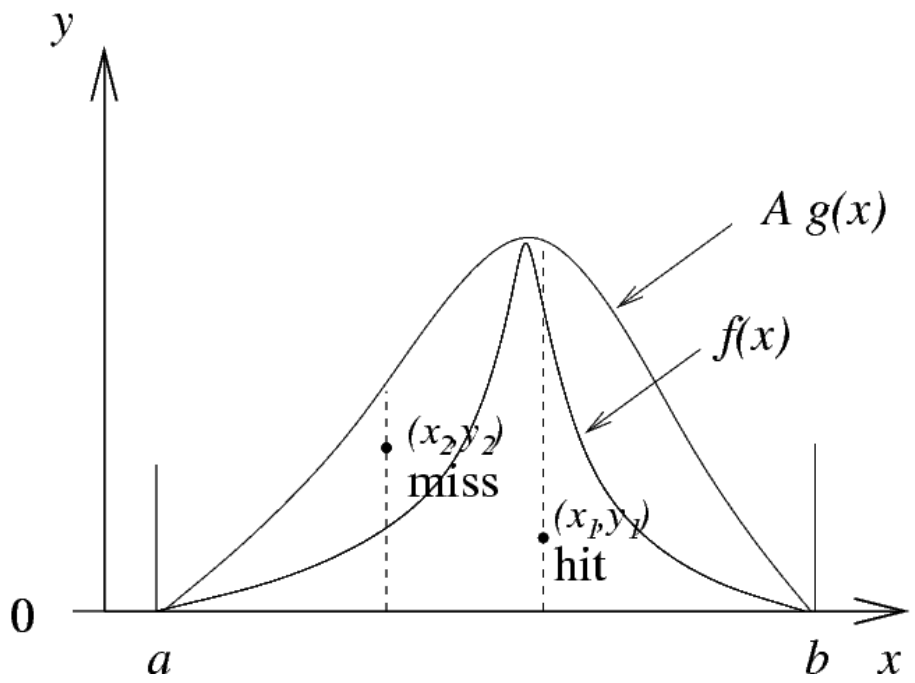
This way we obtain random numbers x which are distributed according to the given distribution $f(x)$.

This works fine, but in case the area under f is small compared to the box, it will be very inefficient. In this case the **combined analytical-rejection** approach may work better:

Assume we can find a function $g(x)$ for which a constant a exists such that

$$ag(x) \geq f(x) \text{ for all } x \in [a, b].$$

It is important to include the constant a here because both $g(x)$ and $f(x)$ are probabilities normalized to one. We further demand that it is possible to form the inverse of the cumulative function $G^{-1}(x)$ of $g(x)$.



Then the algorithm becomes:

1° Generate a uniformly distributed number $u = P_u(0, 1)$

2° Generate a number distributed as $g(x)$: $x = G^{-1}(u)$

3° Generate a uniformly distributed number $y = P_u(0, ag(x))$

4° If $y > f(x)$ this is a **miss:** return to 1°

5° Otherwise this is a **hit:** return x

This also has the advantage that the interval limits a and b can be infinite.

12.4.3.3. Gaussian random numbers

For the very important special case of Gaussian random numbers, the following approach is the most efficient exact method:

1° Obtain $v_1 = P_u(-1, 1)$ and $v_2 = P_u(-1, 1)$ and $w = v_1^2 + v_2^2$

2° If $w \geq 1$ return to step 1°

3° Calculate $r = \sqrt{-2 \log w}$

4° Calculate

$$\begin{cases} x = rv_1/\sqrt{w} \\ y = rv_2/\sqrt{w} \end{cases}$$

This gives a pair of Gaussian deviates (x, y). On the first call the subroutine should return x, on the second call y.

12.4.3.4. Generating points uniformly on a sphere

The correct way to generate points uniformly on a sphere in 3D is:

$$\begin{cases} \theta = \cos^{-1}\left(1 - 2P_u(0,1)\right) \\ \phi = 2\pi P_u(0,1) \end{cases}$$

with the spherical coordinates (θ, ϕ) specifying the location of the point. This also is the correct way of selecting a random direction in 3D.

12.4.4. Stratified sampling and quasi-random distributions

• Stratified sampling: divide the space into a grid, select exactly 1 point randomly in each grid interval.

• Partially stratified sampling: divide the space into a grid, select $N > 1$ points randomly in each grid interval.

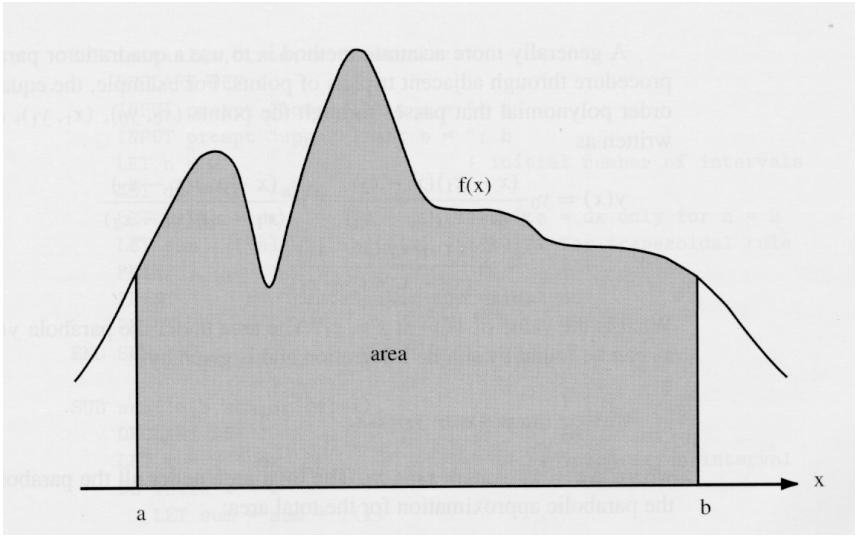
These have the disadvantage that the simulation can only be stopped when a loop over the grid regions has been completed, unless a shuffle operation is used to select the boxes in random order.

• Sobol sequences: generates a sequence of quasi-random numbers which fill space evenly.

All of these methods may give much better convergence in MC integration than ordinary random numbers.

12.5. MC integration

12.5.1. Sampling method

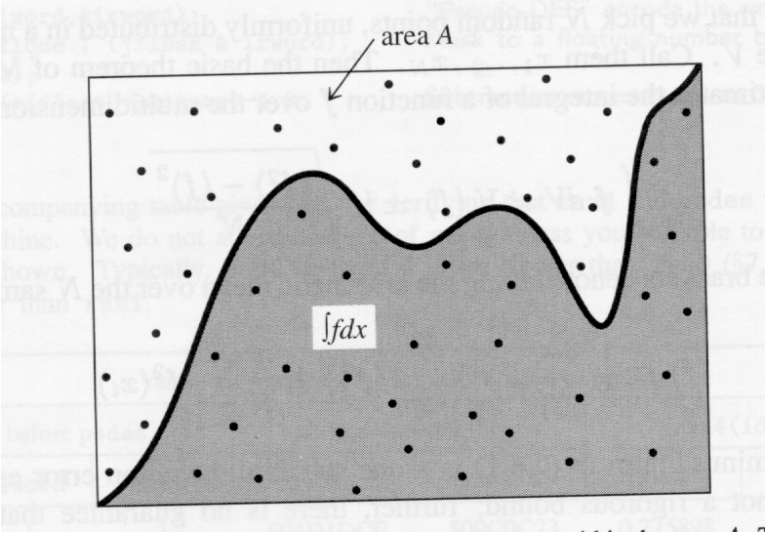


The recipe is simply to pick points $\mathbf{x}$ randomly in the interval of interest, calculate the function value $f(\mathbf{x})$ for each point, and get the average. The integral and its (Gaussian 1 σ) uncertainty is

$$\int f dV \approx V\langle f\rangle \pm V\sqrt{\frac{\langle f^2\rangle - \langle f\rangle^2}{N}} \tag{2}$$

One can also use non-rectangular integration areas, by using random numbers generated in non-uniform distributions to generate the points.

12.5.2. Hit-and-miss (shotgun) method



Here we "shoot with a shotgun" everywhere in an interval completely enclosing the area to be integrated. The answer is just the volume of the enclosing area times the fraction of hits:

$$V = V_e \frac{N_h}{N} \pm V_e \frac{\sqrt{N_h - N_h^2/N}}{N} \tag{3}$$

One can also use non-rectangular enclosing areas, by using random numbers generated in non-uniform distributions to generate the points.

12.5.3. Ways to speed up MC integration

12.5.3.1. Importance sampling

If we want to integrate a function f, and have a function g for which G^{-1} can be calculated, such that

$$\frac{f(x)}{g(x)}$$

is fairly flat, then the convergence of the integral can probably be improved on by calculating the integral as

$$I = \frac{1}{N}\sum_{i=1}^{N}\frac{f(G^{-1}(r_i))}{g(G^{-1}(r_i))}$$

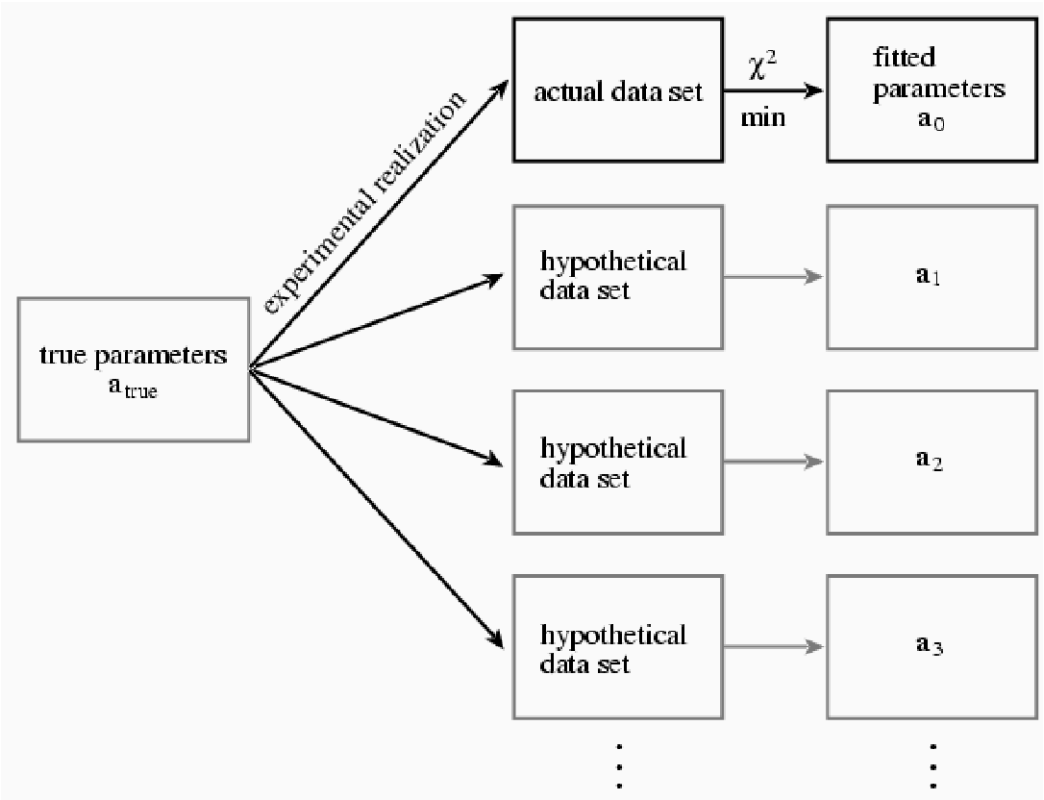
An alternative tool, control variates, is to use a g for which the integral is known, and for which $f(x) - g(x)$ is fairly flat. Then the convergence might be speeded up by MC integrating

$$I = \int_a^b = \int_a^b (f(x) - g(x))dx + \int_a^b g(x)dx.$$

12.5.3.2. Stratified sampling

The ways to generate random numbers in a stratified manner can possibly speed up convergence of MC integration dramatically.

12.6. MC simulation of experimental data



If we only have one original data distribution, but want to deduce the error (and possibly other properties as well), we can use MC to generate **hypothetical** or **synthetic** data sets. Then comparing

the real and hypothetical data sets, one can validly deduce some additional information from the data.

The "original" data set may either be a distribution of known shape, or a single measured set of data. The latter case is called the **bootstrap** method, and should be used with caution.

MC can also be used to simulate the actual process which produces the data. This requires good knowledge of both the physical process, and the measurement which gives out the data.

12.7. Random walks

A "pure" random walk is simply an object moving in random directions, and possibly random distances, either in an M-dimensional lattice or an M-dimensional continuum.

In case the step distance is constant l, then for both continuum and lattice walks, the mean square distance traveled after $N >> 1$ steps is

$$\langle R^2(N) \rangle = l^2 N$$

This can be written into a physically more familiar form by saying that the time between steps is a constant Δt, in which case the time after N steps is

$$t = N \Delta t$$

and if we identify the diffusion coefficient as

$$D = \frac{l^2}{2M\Delta t}$$

we get the Einstein relation

$$\langle R^2(N) \rangle = 2MDt.$$

12.7.1. Other random walks

- **biased** random walks: the probability to move in different directions are not equal

- **persistent** random walks: the jump probability depends on previous history

- **restricted** random walks: the system has special positions where a walker e.g. gets trapped, created or which it can not enter (a block)

- a **lattice gas** is a variety of restricted walks: there the walkers themselves act as blocks for other walkers.

- **self-avoiding** walks: this can also be considered a special kind of restricted walk. Here each walkers is restricted from entering a space where it has previously been.

12.8. Kinetic Monte Carlo

Kinetic Monte Carlo is a random walk either on a lattice or continuum which works for activated processes, and where the real time scale can be calculated.

A simple general form of the KMC algorithm is

0° Set the time $t = 0$

1° Form a list of all the rates r_i of all possible transitions W_i in the system

2° Calculate the cumulative function $R_i = \sum_{j=1}^{i} r_j$ for $i = 1, \ldots, N$ where N is the total number of transitions. Denote $R = R_N$

3° Get a uniform random number $u \in [0, 1]$

4° Find the event to carry out i by finding the i for which

$$R_{i-1} < uR \leq R_i$$

5° Carry out event i

6° Find all W_i and recalculate all r_i which may have changed due to the transition

7° Get a new uniform random number $u \in [0, 1]$

8° Update the time with $t = t + \Delta t$ where:

$$\Delta t = -\frac{\log u}{R} \tag{4}$$

9° Return to step 1

This is $O(N)$. It can in many cases be modified to be below $O(N)$ using data structures and algorithms.

12.9. Thermodynamic MC

The idea of all thermodynamic MC methods discussed on this course is that we do a random walk over phase space points $\mathbf{\Gamma}$, which produces states with a distribution which equals to the probability distribution of the given ensemble

$$\rho_{\rm ens}(\mathbf{\Gamma})$$

When we have such an algorithm, we can calculate a thermodynamic average for a given quantity $\mathcal{A}$ using

$$\langle \mathcal{A} \rangle_{\rm ens} = \frac{\sum\limits_{\substack{\mathbf{\Gamma} \\ \text{chosen from} \\ \text{weight function } \rho}} A(\mathbf{\Gamma})}{N_{\rm MCsteps}}$$

This is a variety of importance sampling.

In all MC methods, particle velocities are not treated explicitly. The velocity-dependent contribution to the desired quantities can (if needed) be added afterwards from the known ideal gas properties.

12.9.1. NVT ensemble

The Metropolis algorithm produces points in phase space distributed according to the canonical ensemble (NVT) weight function, which is just a Boltzmann distribution

$$\rho \propto e^{-V/kT}$$

The algorithm for N particles at positions $\mathbf{r}_i$ is, given a potential energy function $V(\mathbf{r})$:

0 a° Place the particles in some initial configuration $\mathbf{r}_i$.

0 b° Choose a maximum displacement vector (d_x, d_y, d_z)

0 c° Set the number of Monte Carlo steps $n_{MCS} = 0$ and $A_{\rm sum} = 0$

1° Choose a particle i at random among the N particles

2° Calculate the energy of the system before the transition $E_b = V(\mathbf{r})$

3° Generate three uniform random numbers u_1, u_2, u_3 **between -1 and 1**

4° Displace atom i by $\Delta\mathbf{r} = (u_1 d_x, u_2 d_y, u_3 d_z)$: $\mathbf{r}_i = \mathbf{r}_i + \Delta\mathbf{r}$

5° Calculate the energy of the system after the transition $E_a = V(\mathbf{r})$

6° Calculate $\Delta E = E_a - E_b$, then :

- **7°** If $\Delta E \leq 0$ accept the state
- **8°** If $\Delta E > 0$:
  - **8 a°** Generate a random number u between 0 and 1
  - **8 b°** Accept the state only if $u < e^{-\Delta E/kT}$

9° If the state is rejected, return to the previous state: $\mathbf{r}_i = \mathbf{r}_i - \Delta\mathbf{r}$

10° Sum up the desired physical property A for positions $\mathbf{r}$: $A_{\rm sum} = A_{\rm sum} + A(\mathbf{r})$

11° Set $n_{\rm MCS} = n_{\rm MCS} + 1$

12° If $n_{\rm MCS} < n_{\rm max}$ return to step 1

13° Calculate and print out the desired average $\dfrac{A_{\rm sum}}{n_{\rm max}}$

12.9.2. NVE ensemble

The constant-energy microcanonical ensemble can be simulated with the **demon** algorithm, which produces states with constant potential energy V except for a minor fluctuation which is of the order $1/N$.

The algorithm is almost identical to Metropolis. Hence here only the parts which differ from Metropolis are written out.

0 a-c° *As Metropolis.*

0 d° Set the demon energy $E_D = 0$.

1-6° *As Metropolis*

7° If $\Delta E \leq 0$ accept the state and **give the energy ΔE to the demon, $E_D = E_D - \Delta E$**

8° If $\Delta E > 0$:

8 a° If $E_D \geq \Delta E$ let the demon give the necessary energy to the system, $E_D = E_D - \Delta E$, and accept the configuration

8 b° Otherwise the configuration is rejected.

9-13° *As Metropolis*

12.10. Simulated annealing

Simulated annealing means using the Metropolis method starting from a high temperature and gradually lowering it, to minimize the energy of a system.

Simulated annealing can be used to minimize the value of any function f returning a scalar value for a multidimensional configuration space $\mathbf{x}$. The generalized temperature T is just a scalar with the same dimensions as f.

The simulated annealing algorithm can be written as follows; here we include the temperature lowering scheme, and the numbering is different from the main text.

0 a° Select the initial configuration $\mathbf{x}$.

0 b° Set the number of Monte Carlo steps $n_{MCS} = 0$

1° Set $T = T(n_{MCS})$

2° Choose a transition $\Delta\mathbf{x}$ at random

3° Calculate the function value before the transition $f_b = f(\mathbf{x})$

4° Do the trial transition as $\mathbf{x} = \mathbf{x} + \Delta\mathbf{x}$

5° Calculate the function value after the transition $f_a = f(\mathbf{x})$

6° Calculate $\Delta f = f_a - f_b$, then :

- **7°** If $\Delta f \leq 0$ accept the state
- **8°** If $\Delta f > 0$:
  - **8 a°** Generate a random number u between 0 and 1
  - **8 b°** Accept the state only if $u < e^{-\Delta f/T}$

9° If the state is rejected, return to the previous state: $\mathbf{x} = \mathbf{x} - \Delta\mathbf{x}$

10° Set $n_{\mathrm{MCS}} = n_{\mathrm{MCS}} + 1$

11° If $n_{\mathrm{MCS}} < n_{\max}$ return to step 1

$T(n_{MCS})$ should be a function which goes slowly, but not necessarily monotonically, from a high T_0 at $n_{MCS} = 0$ to 0 at $n_{\max}$.

12.11. Cellular automata

Cellular automata work in a discrete "universe" with its own (discrete) set of rules and time which determine how it behaves. Following the evolution of this, often highly simplified, universe, hopefully enables better understanding of our own.

A formal definition can be stated as follows.

1° There is a discrete, finite site space $\mathbf{G}$

2° There is a discrete number of states each site can have $\mathbf{Q}$

3° Time is discrete, and each new site state at time $t + 1$ is determined from the system state $\mathbf{G}$ (t)

4° The new state at $t + 1$ for each site depends only on the state at t of sites in a local neighbourhood of sites V

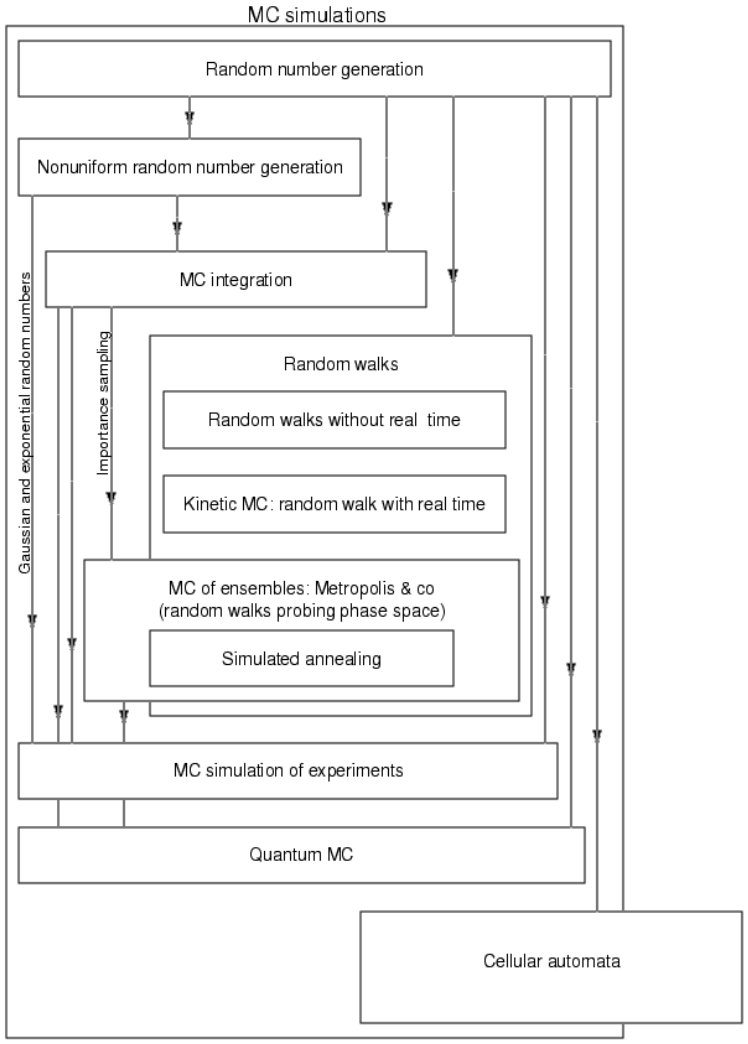
5° There is a rule f which determines the new state based on the old one

Cellular automata can be used to study a wide range of problems in science, ranging from stock market behaviour, evolution and forest fires to fluid flow.

Cellular automata or often not counted as part of MC simulations, but considered a simulation variety of their own. But they are often at least initialized with random numbers, in which case they can be considered MC.

12.12. Overview of important dependencies

The main interrelations between the simulation types describe on this course can be summarized as follows.


MC simulations
Random number generation
Nonuniform random number generation
MC integration
Gaussian and exponential random numbers
Importance sampling
Random walks
Random walks without real time
Kinetic MC: random walk with real time
MC of ensembles: Metropolis & co
(random walks probing phase space)
Simulated annealing
MC simulation of experiments
Quantum MC
Cellular automata

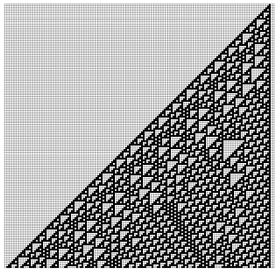
12.13. Final comments

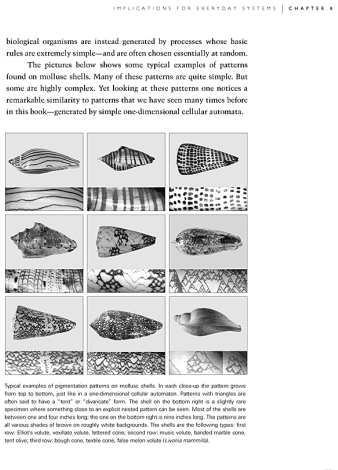
We have seen during this course how a multitude of computational physics methods can be used to treat the same problem. For instance, both the KMC, Metropolis MC, demon MC, and cellular automaton approach can be used to study the Ising model. Similarly, we saw that amazingly simple cellular automata can produce complex fractal shapes, which clearly resemble shapes in nature.

This brings us to an important *caveat*. We must keep in mind that what we see in simulation may not necessarily teach us anything about nature. In an extreme case, the fact that we can reproduce something in nature may be pure coincidence. In this case, it is of no use for us physicists (although it may still be of interest to mathematicians and computer scientists).

As an example of an intermediate case, think about the cellular automata we discussed during the course. Even the 1D automata can produce very nice patterns, like the one on the left for rule 110.

This pattern, and other CS patterns, clearly resemble some of those found on mollusc shells in nature, right side:





IMPLICATIONS FOR EVERYDAY SYSTEMS | CHAPTER 8

biological organisms are instead generated by processes whose basic rules are extremely simple—and are often chosen essentially at random.

The pictures below shows some typical examples of patterns found on mollusc shells. Many of these patterns are quite simple. But some are highly complex. Yet looking at these patterns one notices a remarkable similarity to patterns that we have seen many times before in this book—generated by simple one-dimensional cellular automata.

Typical examples of pigmentation patterns on mollusc shells. In each close-up the pattern grows from top to bottom, just like in a one-dimensional cellular automaton. Patterns with triangles are often said to have a "tent" or "divaricate" form. The shell on the bottom right is a slightly rare specimen where something close to an explicit nested pattern can be seen. Most of the shells are between one and four inches long; the one on the bottom right is nine inches long. The patterns are all various shades of brown on roughly white backgrounds. The shells are the following types: first row: Elliot's volute, vexillate volute, lettered cone; second row: music volute, banded marble cone, tent olive; third row: bough cone, textile cone, false melon volute (*Livonia mammilla*).

423

The crucial question is, does this teach us anything?

The observation of complex patterns arising from simple rules in CA's certainly demonstrates one very important qualitative feature: there is no need for a complex design scheme to achieve the beautiful shapes on the molluscs. They can arise from just a few very simple "rules" determining how the living cells (or whatever) interact during the shell growth.

But this may be all we can learn about the real system. It might be that the rule which gives the outcome actually is related in some way to the interactions between the real cells. But since many different rules can give the same shapes, this is not all that likely. Hence we need better

understanding of the biological interactions before we can use CA's to deduce anything more about the real system.

As another example, consider the case of the annealing of Cu we dealt with on the course. We saw that the Metropolis method is a valid tool for simulating the NVT ensemble. This means in practice that if we from the NVT simulations derive e.g. the heat capacity, we can indeed obtain a value from the simulations which can be compared with experiments. So here the simulations are definitely useful.

But consider then the case were we used Metropolis-based simulated annealing to find the ground state of Cu. Such a simulation (if applied to a less well understood system) could indeed be useful if one wants to find the energy ground state and compare this with experiments. But say we would also want to understand how the actual annealing proceeds in reality. We could from the MC runs derive the system average energy and typical structure at any temperature, and follow their time evolution.

The temperature-dependent energy and structure should in principle be comparable to experiments as well. But if we want to look at the time evolution of how the system evolves, we are in trouble. This is because the Metropolis trial jumps do not correspond directly to any real motion, and there is no time scale. Hence the "Metropolis dynamics" may not have any relation to the real atom dynamics.

Hence being computational scientists, we must keep in mind that we have to understand the basic physics of a system in nature well, before we can be sure whether a computer simulation really can teach us anything about the system studied.